Relaxed Logging for Replay of Multithreaded Applications

(extended abstract of the MSc dissertation)

Aliaksandra Sankova
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—The advent of multi-core processors brought new opportunity to exploit parallelism in programs. However, developing concurrent programs is a sophisticated task, due the new type of bugs that may appear and that may be very difficult to find and to correct. In particular, when multiple threads access shared memory, a buggy parallel program can lead to subtle data races that generate errors in some particular thread interleavings. There is a growing interest on building tools that help to reproduce such interleavings, helping the programmer to correct the code. One of the main techniques to achieve this goal is what has been called *record and replay*: it consists of logging relevant information during the execution of a program that allows the interleaving that causes the bug to be reproduced later. This technique solves the problem of bug reproduction but, unfortunately, in many cases it introduces a substantial slowdown in the execution of the application. This dissertation presents a study of existing approaches to record and replay, reflects on trade-offs and decisions of each system, and proposes a new approach of relaxed logging that aims at reducing the cost of the record phase without introducing a substantial increase in the time required to execute replay phase. We have implemented a prototype to validate these ideas and have evaluated it using several benchmarks.**

I. Introduction

In this dissertation we address the problem of reproducing bugs in concurrent programs that rely on the shared memory paradigm. These programs can be characterized by different sequential threads that execute in parallel, quite often in different cores of a multi-processor, communicating with each other by reading and writing in shared variables. In order to coordinate and communicate among each other, such threads need to use explicit synchronization, such as *locks* or *semaphores*[1]. In particular, when accessing shared data, the logic of the program must ensure that the threads use the required synchronization to avoid *data races*[2]. A data race occurs when different threads access a shared data structure without synchronization, and at least one of those accesses is a write. Data races can be avoided by the correct use of synchronization primitives however these primitives are hard to master. An incorrectly placed or missing synchronization primitive may not only eliminate the race but even create other bugs, for example introduce *deadlocks*[3]. Furthermore, the interleavings that cause the bug may happen only in some rare circumstances, and be very hard to reproduce. This makes the debugging of concurrent programs an extremely difficult and tedious task. In this context we studied techniques and tools that simplify the reproduction of concurrency bugs and came up with an idea of how the state of the art could be improved. Delving into the topic, one of the main techniques to achieve the reproduction of concurrency bugs is what has been called *record and replay*[4], [5], [6], [7], [8], [9], [10], [11], [12], [13] (or *deterministic replay* [14]). Record and replay relies in instrumenting the application in order to record all sources of non-determinism at runtime, including inputs, interrupts, signals, and scheduling decisions. For multi-core environments, it is also necessary to record the order by which different threads have accessed shared variables. This way, deterministic replay can be achieved by re-executing the application while enforcing all points of non-determinism to comply with the information stored in the log. Unfortunately, faithfully logging a concurrent program's execution requires inserting additional synchronization (to ensure that the thread interleaving is correctly traced). This, in combination with the large amount of information that may be required to be captured, can induce an unacceptable slowdown in the application[5], [6], [9], [10]. To address this issue, some approaches attempted to reduce the amount of synchronization used to register the log, the amount of information included in the log, or even both. For instance, one can trace only partial information during the production run and then, at replay time, use search techniques to infer thread interleavings that are compliant with the (partial) log information[8], [13]. However, since reducing the amount of information logged hinders the replay determinism, the challenge lies in identifying the best trade-off between recording cost and inference time.

II. Related Work

Record and replay of multithreaded applications (also known as deterministic replay) is a popular technique to reproduce non-trivial bugs, in particular bugs of non-deterministic nature. The goal of this technique is to allow a given execution of a program to be faithfully reproduced, which requires the ability to reproduce not only the inputs provided to the program, but also all non-deterministic events, such as interrupts and thread interleavings. In general, techniques that provide deterministic replay operate in two distinct phases: record (capturing relevant information about original execution) and replay(re-execution of the program in a controlled manner, ensuring that the run reconstructs the original one). Although simple in theory, building a deterministic replay system poses several issues. Some of them are listed:

- Non-deterministic nature of concurrency bugs can, from time to time, make them disappear when specific interleaving does not occur.
- The record phase may induce a non-negligible overhead during the execution of a program.
- When logging information during production runs, security and privacy issues are raised if some collected data is sensitive.

In order to mitigate the impact of mentioned issues, a large diversity of techniques has been proposed to perform record and replay. These techniques are usually classified according to the amount of specialized support that is built in hardware for this particular purpose. Thus, we can distinguish *hardware-only* systems [15], [16], *hybrid* systems[17] and *software-only* systems[5], [6], [4], [7], [18], [19], [10], [11], [13], [10], [9], [8], [12]. Systems with usage of hardware are able to provide deterministic replay with significantly lower recording overhead than in software solutions. However, bringing these concepts to reality requires expensive non-commodity hardware extensions and ties the approach to a particular architecture. Software-only systems are not as efficient as hardware ones, but are the most common tendency in research of deterministic replay due to their advantage of being more general. In its turn, software-only systems can be classified by the type of information that is recorded in the log. According to [20] it is possible to distinguish three approaches:

- Content-based Approach Such systems are also called data-driven as they record and replay the data read by each instruction. Literally, they log the values of all the relevant inputs and shared variables read by each thread, such that the exact same values can be used during replay. The major drawback of this approach is that it generates very large logs and may induce severe slowdowns in the execution of the program, making approach inefficient [19].
- Order-Based Approach Instead of tracking the values of variables, order-based systems track the control flow of the program (such as timing of interactions with program files or I/O channels) from a given initial state. these types of systems are also called control-driven. According to this approach it is not necessary to record every instruction to replay an execution, which allows to reduce the amount of traced data. However, read and write accesses to shared memory locations still need to be tracked in order to support the reproduction of the thread interleaving. Such tracking is called the *exact linkage* between reads and writes into shared memory. This technique imposes lower overhead on record phase and creates smaller amounts of logs comparing to content-based solutions. The main challenge with this approach is to ensure that the initial state of the program is exactly the same in both the original and the reproduction run. Unfortunately, the initial state may often depend on the availability of external resources such as cores in multicore processors, that could affect the internal state of the program being replayed. Furthermore, to ensure that the log faithfully captures the thread read-write linkage, it is generally necessary to introduce additional synchronization during the record phase. Thus, even if order-based approaches represent an improvement over pure content-based approaches, most systems implemented with this approach still induce a large overhead at runtime [9], [10].
- Search-Based Approach was created with a goal to mitigate an extremely high cost for some applications incurred by assurance of bug reproducibility in every execution, which sometimes would cause up to 100x slowdown [8]. The main idea is to not log the exact thread read-write linkage and instead, to rely on a post-recording phase to construct a feasible interleaving from a *partial log* [9], [8], [12] or to infer the missing information[14]. This way, it is possible to substantially reduce the recording overhead at the cost of smaller determinism guarantees and a potentially longer replay. However, since the search space increases exponentially with the amount of missing information regarding the ordering of thread shared accesses, search-based approaches need to carefully balance the inference time and the recording overhead. Since pure content- and order-based systems incur an overhead is too high to be practical, most recent solutions have adopted a search-based approach. Our solution follows this trend, as well.

In order to shorten the related work chapter, here we only concentrate on systems that were presented within last three years.

1) CARE: CARE [13] is a very recent application-level deterministic record and replay technique for Java concurrent programs. CARE employs an order-based approach that leverages thread locality of variable accesses in order to avoid recording all read-write linkages. Concretely, during the record phase, CARE assigns each thread with a software *cache*. This cache is updated every time the thread reads or writes on a shared variable, and is queried whenever the thread performs a read operation. Write operations are always synchronized and recorded into the trace file whenever a new thread writes on a given shared variable, whereas read operations are only logged in the presence of a *cache miss*. A cache miss occurs when the value read from the shared variable differs from the one previously buffered in the cache, meaning that another thread must have written on this variable before. At this point, CARE logs the exact read-write linkage by redoing the read action again with synchronization.

In the replay phase, CARE does not try to determine all non-recorded read-write linkages. Instead, it simulates the behavior of all caches and overrides read values from memory by values buffered in thread-local caches. This provides value-determinism guarantees at replay. Evaluation shows that CARE resolved all missed linkages for sequentially consistent replay, and exhibited 3.4x reduction on runtime overhead and 7x reduction on log size when compared to LEAP.

Direction of our approach is inspired by two complementary works that explore different points in the design space of record and replay systems, namely STRIDE [12] and CLAP [14].

2) STRIDE [12]: a state-of-the-art search-based solution that targets value determinism by recording bounded shared memory access linkages instead of exact ones. Under the sequential consistency assumption, STRIDE infers a failure-inducing interleaving in polynomial time. Its recording scheme logs read operations without adding extra synchronization, which reduces the runtime overhead with respect to pure order-based approaches [10]. Write operations, on the other hand, are still captured in a synchronized fashion. To allow the reconstruction of the global thread schedule, STRIDE logs read operations with write fingerprints. More concretely, STRIDE associates a version number to all the recorded write operations and, for each read operation, STRIDE records a pair consisting of the value returned by the read operation and the latest version of write that read can possibly link to, i.e., the *bounded linkage*. This bounded linkage is later leveraged by STRIDE's search mechanism to quickly find the correct match between reads and writes. Since STRIDE is the base system of the work in this thesis, we describe its architecture in slightly more detail than the previous systems. The essential concepts of STRIDE are then the *execution log*, the *memory model* and the *legal schedule*. The *execution log* is divided into three parts:

- LW_x, which is a vector containing the local total order of the writes performed by different threads on the shared variable x.
- LA_i, which registers the order of lock/unlock operations on lock l. It allows to reproduce deadlocks.
- TR_i, which corresponds to the read log of thread i.

The *memory model*, in turn, defines the set of values committed by writes that are allowed to be returned by a read. STRIDE uses the most strict memory model for concurrent programs – *sequential consistency* [21]. As defined by Lamport, under sequential consistency, the result of any concurrent execution is the same as if the operations on all the processors are executed in some sequential order and the operations of each individual thread appear in the program order. In this context, the *legal schedule* represents a total order of the read-write operations that conform with the memory behavior rules of STRIDE.

Using these concepts, STRIDE is able to, from an imprecise execution log, generate a feasible thread access ordering such that all read and write operations conform to the sequential consistency memory model. The main advantage of this approach is the low runtime overhead due to the avoidance of additional synchronization when logging reads on shared variables. Authors claim that STRIDE incurs on average of 2.5x smaller runtime slowdown and 3.88x smaller log than LEAP. On the other hand, this approach has the downside of losing some determinism guarantees.

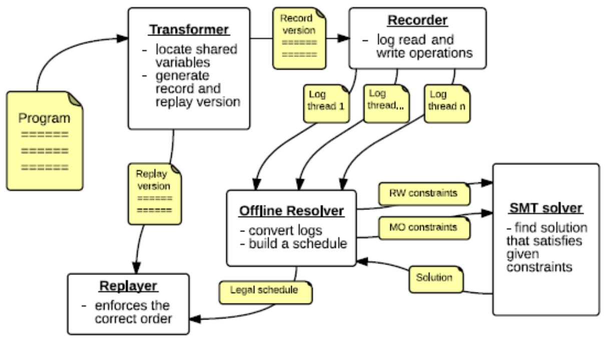

Figure 1. Components of proposed system

3) CLAP [14]: another state-of-the art search-based solution that targets path determinism, created for C/C++ multithreaded programs. Path determinism means this solution does not track values of operations but the path of each thread local execution or, in another words, control flow decisions. Such an approach allows to shrink the search space and gain significant time on the replay phase, as a possibility to take an incorrect execution path be reduced, as will the time of interference. The main idea is to create a constraint model that would represent possible interleavings, encode it as a set of formulae and solve it with help of SMT solver. To further reduce time the solution search time, CLAP generates various candidate schedules to the solver and checks if they satisfy the residuary part of constraints. The main drawback of the approach, used in CLAP is the amount of candidate schedules the system may generate before hitting feasibility. It also produces huge traces, difficult to process with increasing complexity of applications under instrumentation. Thus, depending on programs instrumented, runtime overhead would lay between 9.3% and 269%.

III. OREO

This dissertation introduces OREO, a novel technique for record of replay that aims at exploring a new trade-off between the overhead imposed during the record phase and the time it takes to reproduce the buggy execution during the replay phase. The name of solution stands for "Optimistic approach to REcord phase" of deterministic replay applied to multithreaded applications. The OREO architecture follows the general model of software-only record and replay systems: program is preliminary instrumented, in order to allow to trace information at runtime. The resulting logs are then used to to replay the original run.

Figure 1 illustrates the architecture of our system, which in particular consists of four internal components and one external, namely the *transformer*, the *recorder*, the *offline resolver*, the *replayer* and external *SMT solver*. We describe each component as follows.

- *Transformer:* This component is responsible for the program instrumentation and analysis. It takes the bytecode of an arbitrary Java program and produces two versions: the record version and the replay version. Our transformer role consist of various tasks: i) shared program elements (SPE), i.e. variables that can be concurrently accessed by different threads, ii) collect instructions of access to SPE, iii) collect information about threads. In addition to it, the transformer also instruments two versions of the code: a *record version* and a *replay version*, that will serve as input to the recorder and the replayer, respectively.
- *Recorder:* This component executes the record version of the program (previously instrumented by the *transformer*) and stores the relevant events into a trace file. Concretely, the *recorder* produces per thread logs, containing the write operations, the read operations, the lock acquisition order and information about thread creation. Unlike STRIDE, we do not introduce any kind of synchronization to log write operations, so writes with the same version are allowed.
- *Offline Resolver:* This component processes logs produced by the *recorder*, with the goal of producing a feasible execution ordering of events, i.e., the *legal schedule*. To this end, the offline resolver changed philosophy of its existence, to recall that the name remained unchanged. It first was created to build up a schedule by bounded linkage infer, time to time referring to conflicts resolution. After its role changed, in current system prototype the offline-resolver is used to convert logs from recorder to a set of SMT formulae, consisting of Read-Write constrains that represent local order or order per thread, and Memory Model constraints, which we build with the algorithm, similar to bounded linkage infer from STRIDE. All this we pass to the *SMT solver*
- *SMT Solver (external):* is a high-performance theorem prover that is accessed by a part of code from *offline-resolver* called SMT Connector. It returns a set of SMT formulae describing expressions that satisfy provided constraints, which is further converted to a *legal schedule*.
- *Replayer:* This component controls the scheduling of threads to enforce a deterministic replay using both the access vectors and the thread identity information. In such a way, if there is a bug in execution, it will get triggered in a deterministic way. Thus, developers working with multithreaded applications, gain possibility to repeat the events in their program (so-called cyclic debugging) and get the vision of what is happening among threads.

A. Usage of SMT and Constraint Model

The idea to use SMT constraint solving to replay an execution is not new. For instance, Lee et al. have been experimenting with offline symbolic analysis for deterministic replay at the hardware level[22], [23] as the majority of modern processors operate with relaxed memory model in order to enable performance optimization. However, their solutions are not applicable in our case as they require expensive hardware modifications. Bringing their ideas to software-only systems would mean aiming at value determinism and therefore, feeding the SMT solver with non-trivial symbolic expressions, to find the legal schedule. To circumvent this drawback, CLAP [14] collects per-thread path profiles at runtime and uses them to guide a symbolic execution of the program, collecting symbolic information with respect to data-flow, control-flow and synchronization operations. This way, CLAP is able to build a constraint model that requires the solver just to reason on orderings of operations, instead of having to find the actual values returned by the read operations that allow to satisfy the constraints. However, since CLAP only records the execution path that each thread followed during the production, in the constraint model, it has to generate constraints that match each read on a shared variable to all the writes on that variable. As a result, CLAP exhibits poor scalability. Consequently, let us present the set of constraints that OREO passes to the SMT solver to obtain a legal schedule. Our idea is to use logged values to encode a constraint model representing the feasible thread interleavings that conform with the original execution. To facilitate constraint creation we use a dictionary data structure, where the key is a combination of a shared variable identifier with the value of a write operation. The value itself is a list of all the events that happened to the same field with the same value - a collection of writes with different versions. From above, we encode all the necessary constraints into a formula Φ constructed by a conjunction of four sub-formulae:

$$\Phi = \Phi mo \land \Phi rw \land \Phi wv \land \Phi so \tag{1}$$

where Φmo stands for the memory order constraints, Φrw denotes the read-write constraints over captured SPE accesses, Φwv are constraints which denote that among write operations the earliest has the lowest version number and Φso denotes synchronization order constraints that define the order of thread-related events. We further explain each constraint further.

1) Intra-thread Constraints: The following constraints are built using the information collected in a single per-thraed log, and aim to restrict the order of events within a thread:

- Memory Order Constraints Memory Order Constraints allow us to preserve partial order in which instructions are executed in each thread and represent data flow within the execution. This is important as it allows us to follow the original flow and therefore, reproduce the bug when needed. In order to produce these type of constraints we parse events from the log and sort them according to the value of eventId, per-thread counter of events. Thus, event with a smaller eventID value happened earlier in what corresponds to a sequential consistent memory model.

- Write Versioning Constraints A state where writes should be ordered according to the versions captured at runtime. We again sort events observed within thread execution and sort them by version value so the least version would denote the earliest event. This constraint reduces time of linkage inference when in the global execution different threads managed to perform write operations with the same version.

2) Inter-thread Constraints: These constraints aim to provide global order sequence of events from all the log instances hence adhere to a legal schedule of the original execution with the maximum alignment:

- Read-Write Constraints Read-write constraints represent a linkage between read and write operations across the execution, stating their execution order. Let us consider a read operation R_i on a shared variable where i is a particular version of captured access and W_i is one or more write operation on the same variable. OR_i will denote the order of read and OW_i the order of write operations respectively. To compose read-write constraints we scan the log from the end to the beginning, looking for a corresponding write access that would match in value and in version to each read operation. If a perfect match is found, we add a string denoting that the write operation happened before the read one:

$$OW_1 < OR_1 \tag{2}$$

If there are more write operations with the same version, we take a slightly different approach. Let us consider that W_i^{T1} and W_i^{T2} denote write operations on the same shared variable, which were occasionally captured with the same version (due to data races). There will be two mutually exclusive ways to sort them out:

1) R_i happened after W_i^{T1}. Then either W_i^{T2} had to happen before W_i^{T1} or this local ordering is not feasible.
2) R_i happened after W_i^{T2}. Then either W_i^{T1} had to happen before W_i^{T2} or this local ordering is not feasible.

Adding such branching to will enlarge the search space but also increase probability to actually repeat the original execution. Thus, for this particular situation we will add the following constraint:

$$\begin{aligned}&((R_i < W_i^{T1})\wedge\\&((W_i^{T2} < W_i^{T1}) \vee (R_i < W_i^{T2})))\wedge\\&((R_i < W_i^{T2})\wedge\\&((W_i^{T1} < W_i^{T2}) \vee (R_i < W_i^{T1})))\end{aligned} \tag{3}$$

In such a way, n write operations with the same version will produce $n!$ mutually exclusive ways to match them to each read. Here SMT solver will help us to gain better increase in time of execution than custom algorithm with Java data structures.

- Synchronization Order Constraints Synchronization Order Constraints in OREO aim at enforcing the control flow of the program and therefore, ordering *partial order* events such as START, EXIT, JOIN, FORK while relaxing the model on *locking* and such events as SIGNAL, WAIT. This approach allows us to speed up building a model for Z3 while still being able to determine happens-before relation in general and avoid locking. In such a way, OREO imposes 4 types of partial order constraints:

1) *START* $<$ *EXIT* denotes the thread lifetime. The thread cannot finish before starting.
2) *FORK* $<$ *START* denotes how the child thread should be created.
3) *FORK* $<$ *JOIN* denotes the global rule of child thread creation.
4) *EXIT* $<$ *JOIN* denotes how the child thread should be finished.

Finally, the constraint formulae set is sent to a SMT solver.

B. Our model versus CLAP

Our approach is similar to one proposed in CLAP[14], however it is operating on a simpler memory model. The main differences are listed:

– Simplified read-write: OREO significantly reduces the size of the read-write constraints generated by CLAP, by tracing the value that each read saw at runtime. This way, OREO encodes only the matching of a read with the set of possible writes that wrote that same value.
– Eliminated locking constraints: OREO records the locking order at runtime, which allows to eliminate the locking constraints produced by CLAP altogether. Note that this set of constraints is cubic on the number of locking pairs, which results in a substantial reduction of the solving time. This fact was previously in shown in [24].

We believe that OREO moves a step forward towards the sweet spot between finding a legal schedule inference via constraint solving and recording overhead. Since we trace the exact value that a read operation saw at runtime, we are able to substantially reduce the amount of writes that it can be matched with. As a result, the constraint model built by OREO is simpler than that of CLAP, and the SMT solver is capable of finding a solution more easily. From the other side, conservative record and replay solutions introduce a significant amount of additional synchronization to ensure that accurate information regarding thread interleavings is stored in the log. Recent work proposed a number of techniques to reduce this amount of synchronization. Our work also explores this path: we introduce a relaxation of the logging procedure removing locking from write operations, used in STRIDE. Removing these locks allows us to gain an increase in execution speed. Although tracing the write order without synchronization may cause conflicts in the write versions mechanism (e.g., two writes may be recorded with the same version number), the experiments in [12] showed that the read/write operation and the corresponding recording operation tend to occur contiguously for most programs. Our experiments in Section IV further support this claim.

C. Implementation Details

1) Transformer: The OREO transformer is developed on top of Soot[25], a static program analysis framework for optimizing Java bytecode, developed at McGill University in 1999. This framework supports three intermediate representations for representing Java bytecode: Baf, a streamlined representation of Java's stack-based bytecode; Jimple, a typed three-address intermediate representation suitable for optimization; and Grimp, an aggregated version of Jimple. Our transformer takes the bytecode of the Java program selected for analysis, and performs its instrumentation on Jimple, in particular:

– localizes shared program elements (SPEs), which we recognize in two types: shared variables (static and instance elements of class that can be accessed by different threads), and synchronization variables (locks and monitors).
– produces record and replay code versions by injection of Jimple probes into original code.
– visits each Jimple statement and performs tasks they describe.

Our transformer can instrument only Java programs. Depending on nature of SPE it inserts different probes:

– Operations on shared variables beforeLoad, afterLoad, beforeStore, afterStore;
– Operations on synchronized variables beforeMonitorEnterStatic, afterMonitorEnterStatic, recording calls to *signal()* and *await()* beforeConditionEnter, afterConditionEnter;
– Monitor acquisition operations for instance invocations beforeMonitorEnter, afterMonitorEnter, exitMonitor;
– Thread behavior and control flow events mainThreadStartRun, threadStartRun, threadExitRun, and so on.

Thus, the OREO transformer can be seen as a simplified version of the transformer, proposed by Huang et al.[10]. We inject less instructions, do not provide annotation-based specifications for inserted end-points and therefore, do not support user specified end points on record. Transformer output provides us with the number of shared variables and the number of synchronization variables that are necessary to call record or replay driver.

2) Recorder: When the record version is running, we pass through all injected Jimple probes and collect information about accesses to SPE in internal data structures. To represent each operation, we implemented a java class Event that can exist in 8 following types: read, write, lock, unlock, join, fork, start, exit. For each thread we create a counter, that assigns a partial order position to operations within the thread. Unlike STRIDE, we do not introduce any kind of synchronization to log write operations, so writes with the same version number are allowed. A conflict can be detected when there is more than one write operation on the same shared variable with the same event version number - as we increase version counter on each access, we can see a bug introduced by data races. We also maintain statistics of conflict ratio per run. On every thread fork, the parent thread ID is added to the thread creation order list. In order to save logs, we insert a ShutDownHook into the JVM Runtime in the recorder as an end point. When the program execution is over, the ShutDownHook will be invoked to perform saving of captured traces from internal data structures to physical files. Logs are stored in the JSON format.

3) Offline-Resolver: The SMT Solver used in OREO is the Z3[26], a high-performance theorem prover developed at Microsoft Research that is being used in several projects since February 2007. It is targeted at solving problems that arise in software verification and software analysis. While supporting various theories, it has proven to be useful in such application areas as extended static checking, predicate abstraction, test case generation. In our system we interact with the Z3 through a component written in Java, the Z3Connector, which uses Z3's binary API inside, providing us with a simplified interface to submit formulas in a text format. The reason for using the Z3 in OREO is its ability to produce models as part of the output by assigning values to the constants in the input and generating partial function graphs for predicates and function symbols. This allows us literally return a legal schedule, which requires a small conversion to be executed on replay driver.

4) Replayer: The Replayer controls the scheduling of threads and operations performed by them. Replay is based on information from the access vectors, received from the model provided by Z3, and the thread creation order list. Before creating a new thread, if compares the ID of the parent thread to the ID at the head of the thread creation order list. If they match, the new thread is allowed to be created and the head of the list is removed. Before replaying any access, threads coordinate with each other by semaphores, counting amount of SPE already accessed. Process is quick as the threads accessing different SPEs can execute in parallel.

IV. Evaluation

We conducted experiments with the purpose of evaluating our system according to the following criteria:

- *Recording overhead*, in terms of performance slowdown imposed by the logging, with respect to the native execution.
- *Log size*, in terms of the amount of information stored in the trace files during the production run.
- *Amount of write conflicts*, in order to understand the impact of tracing write operations without synchronization and to assess the likelihood of finding write versioning conflicts that vary with the nature of the application (e.g. the percentage of write operations, the number of threads, and the shared accesses).
- *Inference Time*, to evaluate whether our system will be able find and produce a legal schedule that triggers the concurrency bug in a practical amount of time.

As test subjects, we first used a micro-benchmark, named *Bank*, that simulates (unsynchronized) transfers on bank accounts. As this micro-benchmark allows to easily tune the number of threads and the number of shared accesses, we were able to assess OREO's benefits and limitations with respect to STRIDE in the presence of different execution scenarios.

In addition, we also tested OREO against STRIDE with four other third-party benchmarks: two from the IBM ConTest suite [27] and two from the *Java Grande*[1] suite.

A. Bank Micro-benchmark

In order to evaluate both the performance and the replay ability of OREO, we started by developing a micro-benchmark that allows to easily tune the number of threads and shared accesses used in the experiments. This micro-benchmark consists of an application that simulates transfers between bank accounts. Since the threads concurrently update the accounts with no synchronization, the final balance may not be correct.

An execution of the benchmark corresponds to performing a number num_ops of operations (i.e., transfers between accounts). Here, each thread executes num_ops/num_threads transfers, where num_threads indicates the number of threads executing. Once a thread finishes its transfers, it does a sanity check to verifies whether the sum of all individual accounts is equal to the expected total balance. In case of inconsistency, the thread raises an exception that prints a message indicating that a bug has occurred.

To assess how OREO compares against STRIDE in the presence of different complexity scenarios, we executed the bank micro-benchmark with 12 distinct configurations. These configurations were obtained by

[1]http://www.javagrande.org

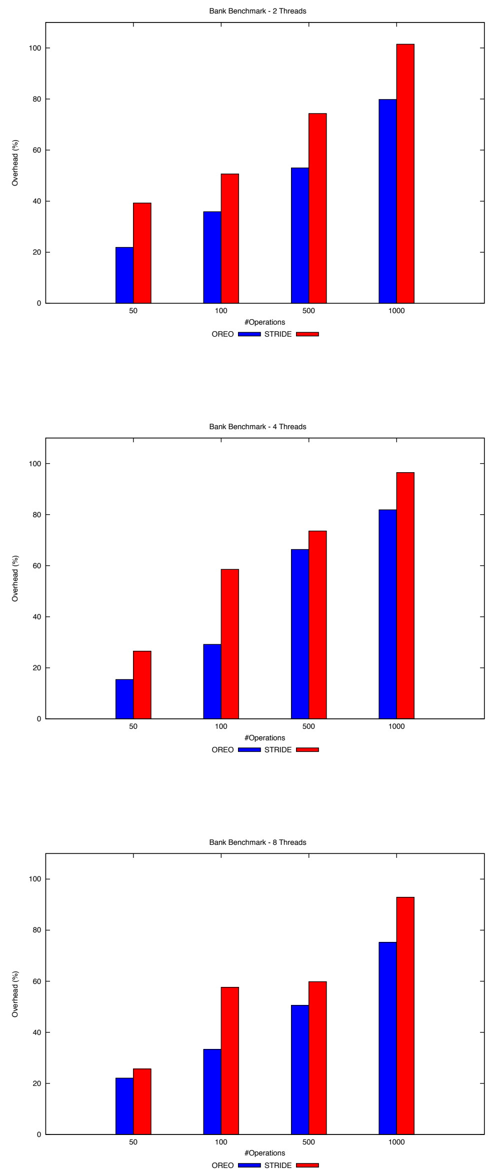

Figure 2. Recording overhead (%) for OREO and STRIDE for benchmark Bank, executed with 2, 4, and 8 threads. Results are averaged over 5 runs.

varying the number of operations of the program for the set of values $\{50, 100, 500, 1000\}$, as well as the number of threads for the set of values $\{2, 4, 8\}$. Finally, the number of accounts (i.e., of shared variables) was set to 10.

1) Recording Overhead: Figure 2 depicts the recording overhead, with respect to the native execution time, of both OREO and STRIDE for the bank micro-benchmark, when varying the number of threads and the number of operations. The plots show that OREO imposed less runtime slowdown than STRIDE for all

	#Operations			
#Threads	**50**	**100**	**500**	**1000**
2	45KB	74KB	305KB	597KB
4	47KB	77KB	307KB	598KB
8	54KB	82KB	311KB	603KB

Table I
LOG SIZES FOR OREO.

	#Operations			
#Threads	**50**	**100**	**500**	**1000**
2	5s	50s	3h56m	>8h
4	23s	4m22s	5h16m	>8h
8	59s	4m1s	7h1m	>8h

Table II
AMOUNT OF TIME REQUIRED TO SOLVE THE CONSTRAINT MODEL WITH THE READ-WRITE LINKAGES AND PRODUCE A LEGAL SCHEDULE.

cases. On average, STRIDE incurred 64% recording overhead, whereas OREO achieved an overhead of 45%, thus being 1.4x faster than STRIDE.

Another interesting observation that we can draw from Figure 2 is that the overhead does not increase linearly with the number of threads. This is because an increasing number of threads implies a smaller of operations executed by thread.

Finally, we expected the overhead reduction achieved by OREO to increase with the number of operations, as the negative effect of using locks would be more visible. We believe that this did not happen in this benchmark because the amount of writes did not comprise the majority of the operations of the program. As a result, the negative impact of STRIDE's additional synchronization ended to be diluted in the overall instrumentation cost, required to log the events.

2) Log Sizes: Since our implementations of OREO and STRIDE share the same data structures and event objects, the trace files generated by the two techniques are identical. As such, we solely report the log sizes for OREO. Table I reports these results.

As expected, Table I shows that the size of the logs increase with the number of operations performed during the execution. Also, despite being the same amount of operations, having more threads also results in slightly larger logs, because each thread has a dedicated array where it stores its events. Hence, there is a small fixed space cost resulting from these per-thread data structures.

3) Write Conflicts: Curiously, for all the experiments, we have never observed version conflicts when logging write operations without additional synchronization. This provides further support our initial claim that version conflicts are rare.

4) Inference Time: Table II reports the time OREO took to produce a legal schedule, corresponding to the solving time of the constraint model. It is possible to see that an increase in the number of operations has a significant impact on the solving, as the SMT solver required 10x more time to solve the model when doubling the amount of operations from 50 to 100. Moreover, for the cases where the benchmark was configured to perform 1000 operations, the solver took even more than 8h. This indicates that for long executions, it might be useful to perform a first stage when read-write linkages are resolved solely using write versions. Then, in case of version conflicts, one could resort to the SMT solver to produce the legal schedule.

As a final remark, we highlight the fact that OREO was able to reproduce the bugs in the different executions using the inferred schedules.

B. Third-Party Benchmarks

In order to further assess OREO's benefits and limitations, we also conducted experiments with four third-party benchmarks. We used the benchmark bugs *TwoStage* and *Airline* from the IBM ConTest benchmark suite [27], and programs *Crypt* and *LUFact* from the Java Grande benchmark[2] (both with input size B). Table III reports the results of our experiments. Columns 1-3 characterize the programs in terms of their number of threads and shared accesses. Columns 4 and 5 report the recording overhead for OREO and STRIDE, respectively. Column 6 shows the log size generated by OREO. Column 7 indicates the number of write conflicts observed during the execution and, finally, column 8 contains the time required to solve the constraint model.

1) Recording Overhead: Comparing the recording overhead of OREO to that of STRIDE for the third-party benchmarks, it is possible to see that, once again, OREO outperforms STRIDE for all test cases. The most prominent case is *TwoStage*, where OREO was 8.7x faster than STRIDE, reducing the recording overhead from 41.7% to 4.8%. This program has only 41 shared accesses and has some synchronization operations on the original code as well, which caused OREO's instrumentation to not be too harmful for the program's performance. On the other hand, since STRIDE injects extra locks to record write operations, it imposed a non-negligible slowdown at runtime.

2) Log Sizes: The log sizes produced by OREO (which are identical to those of STRIDE) ranged from 3.1KB for *TwoStage* to 373KB for *LUFact*. As expected, the logs were larger for programs with more shared accesses, such as *LUFact*.

We stored the traces as strings in JSON format, to ease user readability. Therefore, we believe that the space overhead could be reduced by using more efficient data structures, such as serializable objects, and compression techniques.

3) Write Conflicts: Similarly to the previous section, we did not observe any write conflicts on the third-party

[2]http://www.javagrande.org

Program	#Threads	#Shared Accesses	Recording Overhead (%)		Log Size	#Write Conflicts	Solving Time
			OREO	STRIDE			
TwoStage	5	41	4.8	41.7	3.1KB	0	0.13s
Airline	11	94	13.7	43.5	7.5KB	0	1.93s
Crypt	3	57	1.3	10.6	4KB	0	0.23s
LUFact	2	4037	42.5	64.3	373KB	0	29m

Table III
RESULTS FOR THIRD-PARTY BENCHMARKS.

benchmarks. Curiously, this did not even occur in the *Airline* program, which had 11 threads running. We believe that this is due to the fact that, despite having many concurrent threads, *Airline* did not have many shared write operations to incur version conflicts.

4) Inference Time: Regarding inference time, we can see in Table III that OREO is able to find the read-write linkage in a very short amount of time for programs *TwoStage*, *Airline*, and *Crypt*. For *LUFact*, however, OREO took 29m to produce a legal schedule, due to the large number of shared accesses in this program.

To further compare OREO against prior work, we generated the CLAP's constraint model for the *LUFact* test case and tried to solve it[3]. The solver found a solution in 1h, which was twice the time it took for OREO.

C. Summary of Evaluation

The results from our experiments showed that OREO incurs, on average, 1.4x and 2.6x less recording overhead than STRIDE, respectively for the Bank micro-benchmark and the third-party benchmarks. In terms of constraint solving, OREO was also 2x faster than CLAP to resolve the read-write linkages for the program *LUFact*. Finally, OREO was able to produce a legal schedule, capable of replaying the original execution, within 7 hours for most test cases, which we argue to be a practical amount of time to be used in debugging.

V. CONCLUSIONS

This dissertation has presented and evaluated OREO, a novel approach to record-and-replay. OREO aims at making the logging phase more efficient, using a relaxed logging procedure and offline algorithms to link operations and produce a replay schedule. OREO combines features of different state-of-the-art systems, such as STRIDE[12] and CLAP[14] to get better results. We have implemented OREO and performed an experimental evaluation assess its advantages and limitations.

For future work we would like to explore adaptive mechanisms in the record phase, to switch between exact and relaxed write logging, thus allowing to automatically tune the recording scheme to an application's nature.

[3]We experimented with *LUFact* alone, as this was the program for which the constraint model exhibit more complexity and the solver had required more time to find a solution.

ACKNOWLEDGMENTS

This project has been funded with support from the European Commission. This report reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Nuno Machado and Manuel Bravo, to whom I express my deepest gratitude for their support and guidance.

REFERENCES

[1] E. W. Dijkstra, *Cooperating sequential processes*. Springer, 2002.

[2] R. Netzer and B. P. Miller, *Detecting data races in parallel program executions*. University of Wisconsin-Madison, Computer Sciences Department, 1989.

[3] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, 1971.

[4] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. ACM, 1998, pp. 48–59.

[5] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere, "Jarec: a portable record/replay environment for multi-threaded java applications," *Software: practice and experience*, vol. 34, no. 6, pp. 523–547, 2004.

[6] M. Ronsse and K. De Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 133–152, 1999.

[7] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *Computers, IEEE Transactions on*, vol. 100, no. 4, pp. 471–482, 1987.

[8] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 177–192.

[9] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 193–206.

[10] J. Huang, P. Liu, and C. Zhang, "Leap: lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 207–216.

[11] Z. Yang, M. Yang, L. Xu, H. Chen, and B. Zang, "Order: Object centric deterministic replay for java." in *USENIX Annual Technical Conference*, 2011.

[12] J. Zhou, X. Xiao, and C. Zhang, "Stride: Search-based deterministic replay in polynomial time via bounded linkage," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 892–902.

[13] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "Care: cache guided deterministic replay for concurrent java programs." in *ICSE*, 2014, pp. 457–467.

[14] J. Huang, C. Zhang, and J. Dolby, "Clap: recording local executions to reproduce concurrency failures," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 141–152.

[15] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 289–300.

[16] M. Xu, R. Bodik, and M. D. Hill, "A" flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 122–133.

[17] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 73–84.

[18] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 97–108, 2009.

[19] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 154–163.

[20] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. Bosschere, "A taxonomy of execution replay systems," in *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*. Citeseer, 2003.

[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[22] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline symbolic analysis for multiprocessor execution replay," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 564–575.

[23] D. Lee, M. Said, S. Narayanasamy, and Z. Yang, "Offline symbolic analysis to infer total store order," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 357–358.

[24] M. Bravo, N. Machado, P. Romano, and L. Rodrigues, "Towards effective and efficient search-based deterministic replay," in *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, ser. HotDep '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:6.

[25] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[26] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[27] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS'03. IEEE Computer Society, 2003, pp. 286–293.